05-SpringDI容器：如何分析一个软件的模型？

你好！我是郑晔。

在上一讲中，我们讨论了如何了解一个软件的设计，主要是从三个部分入手：模型、接口和实现。那么，在接下来的三讲中，我将结合几个典型的开源项目，告诉你如何具体地理解一个软件的模型、接口和实现。

今天这一讲，我们就先来谈谈了解设计的第一步：模型。如果拿到一个项目，我们怎么去理解它的模型呢？

我们肯定要先知道项目提供了哪些模型，模型又提供了怎样的能力。这是所有人都知道的事情，我并不准备深入地去探讨。但如果只知道这些，你只是在了解别人设计的结果，这种程度并不足以支撑你后期对模型的维护。

在一个项目中，常常会出现新人随意向模型中添加内容，修改实现，让模型变得难以维护的情况。造成这一现象的原因就在于他们对于模型的理解不到位。

我们都知道，任何模型都是为了解决问题而生的，所以，理解一个模型，需要了解在没有这个模型之前，问题是如何被解决的，这样，你才能知道新的模型究竟提供了怎样的提升。也就是说，**理解一个模型的关键在于，要了解这个模型设计的来龙去脉，知道它是如何解决相应的问题。**

今天我们以Spring的DI容器为例，来看看怎样理解软件的模型。

耦合的依赖

Spring在Java世界里绝对是大名鼎鼎，如果你今天在做Java开发而不用Spring，那么你大概率会被认为是个另类。

今天很多程序员都把Spring当成一个成熟的框架，很少去仔细分析Spring的设计。但作为一个从0.8版本就开始接触Spring的程序员，我刚好有幸经历了Spring从渺小到壮大的过程，得以体会到Spring给行业带来的巨大思维转变。

如果说Spring这棵参天大树有一个稳健的根基，那其根基就应该是 Spring的DI容器。DI是Dependency Injection的缩写，也就是“依赖注入”。Spring的各个项目都是这个根基上长出的枝芽。

那么，DI容器要解决的问题是什么呢？它解决的是**组件创建和组装**的问题，但是为什么这是一个需要解决的问题呢？这就需要我们了解一下组件的创建和组装。

在前面的课程中，我讲过，软件设计需要有一个分解的过程，所以，它必然还要面对一个组装的过程，也就是把分解出来的各个组件组装到一起，完成所需要的功能。

为了叙述方便，我采用Java语言来进行后续的描述。

我们从程序员最熟悉的一个查询场景开始。假设我们有一个文章服务（ArticleService）提供根据标题查询文章的功能。当然，数据是需要持久化的，所以，这里还有一个ArticleRepository，用来与持久化数据打交道。

熟悉DDD的同学可能发现了，这个仓库（Repository）的概念来自于DDD。如果你不熟悉也没关系，它就是

与持久化数据打交道的一层，和一些人习惯的Mapper或者DAO（Data Access Object）类似，你可以简单地把它理解成访问数据库的代码。

```
class ArticleService {
  //提供根据标题查询文章的服务
  Article findByTitle(final String title) {
    ...
  }
}

interface ArticleRepository {
  //在持久化存储中，根据标题查询文章
  Article findByTitle(final String title);
}
```

在ArticleService处理业务的过程中，需要用到ArticleRepository辅助它完成功能，也就是说，ArticleService要依赖于ArticleRepository。这时你该怎么做呢？一个直接的做法就是在 ArticleService中增加一个字段表示ArticleRepository。

```
class ArticleService {
  private ArticleRepository repository;

  public Article findByTitle(final String title) {
    // 做参数校验
    return this.repository.findByTitle(title);
  }
}
```

目前看起来一切都还好，但是接下来，问题就来了，这个字段怎么初始化呢？程序员一般最直接的反应就是直接创建这个对象。这里选用了一个数据库版本的实现（DBArticleRepository）。

```
class ArticleService {
  private ArticleRepository repository = new DBArticleRepository();

  public Article findByTitle(final String title) {
    // 做参数校验
    return this.repository.findByTitle(title);
  }
}
```

看上去很好，但实际上DBArticleRepository并不能这样初始化。正如这个实现类的名字所表示的那样，我们这里要用到数据库。但在真实的项目中，由于资源所限，我们一般不会在应用中任意打开数据库连接，而是会选择共享数据库连接。所以，DBArticleRepository需要一个数据库连接（Connection）的参数。在这里，你决定在构造函数里把这个参数传进来。

```
class ArticlService {
  private ArticleRepository repository;

  public ArticlService(final Connection connection) {
    this.repository = new DBArticleRepository(connection);
  }

  public Article findByTitle(final String title) {
    // 做参数校验
    return this.repository.findByTitle(title);
  }
}
```

好，代码写完了，它看上去一切正常。如果你的开发习惯仅仅到此为止，可能你会觉得这还不错。但我们并不打算做一个只写代码的程序员，所以，我们要进入下一个阶段：测试。

一旦开始准备测试，你就会发现，要让ArticleService跑起来，那就得让ArticleRepository也跑起来；要让ArticleRepository跑起来，那就得准备数据库连接。

是不是觉得太麻烦，想放弃测试。但有职业素养的你，决定坚持一下，去准备数据库连接信息。

然后，真正开始写测试时，你才发现，要测试，你还要在数据库里准备各种数据。比如，要测查询，你就得插入一些数据，看查出来的结果和插入的数据是否一致；要测更新，你就得先插入数据，测试跑完，再看数据更新是否正确。

不过，你还是没有放弃，咬着牙准备了一堆数据之后，你突然困惑了：我在干什么？我不是要测试服务吗？做数据准备不是测试仓库的时候应该做的事吗？

那么，问题出在哪儿呢？其实就在你创建对象的那一刻，问题就出现了。

分离的依赖

为什么说从创建对象开始就出问题了呢？

因为当我们创建一个对象时，就必须要有一个具体的实现类，对应到我们这里，就是那个DBArticleRepository。虽然我们的ArticleService写得很干净，其他部分根本不依赖于DBArticleRepository，只在构造函数里依赖了，但依赖就是依赖。

与此同时，由于要构造DBArticleRepository的缘故，我们这里还引入了Connection这个类，这个类只与DBArticleRepository的构造有关系，与我们这个ArticleService的业务逻辑一点关系都没有。

所以，你看到了，只是因为引入了一个具体的实现，我们就需要把它周边配套的东西全部引入进来，而这一切与这个类本身的业务逻辑没有任何关系。

这就好像，你原本打算买一套家具，现在却让你必须了解树是怎么种的、怎么伐的、怎么加工的，以及家具是怎么设计、怎么组装的，而你想要的只是一套能够使用的家具而已。

这还只是最简单的场景，在真实的项目中，构建一个对象可能还会牵扯到更多的内容：

- 根据不同的参数，创建不同的实现类对象，你可能需要用到工厂模式。
- 为了了解方法的执行时间，需要给被依赖的对象加上监控。
- 依赖的对象来自于某个框架，你自己都不知道具体的实现类是什么。
- ……

所以，即便是最简单的对象创建和组装，也不像看起来那么简单。

既然直接构造存在这么多的问题，那么最简单的办法就是把创建的过程拿出去，只留下与字段关联的过程：

```
class ArticleService {
  private ArticleRepository repository;

  public ArticleService(final ArticleRepository repository) {
    this.repository = repository;
  }

  public Article findByTitle(final String title) {
    // 做参数校验
    return this.repository.findByTitle(title);
  }
}
```

这时候，ArticleService就只依赖ArticleRepository。而测试ArticleService也很简单，只要用一个对象将ArticleRepository的行为模拟出来就可以了。通常这种模拟对象行为的工作用一个现成的程序库就可以完成，这就是那些Mock框架能够帮助你完成的工作。

或许你想问，在之前的代码里，如果我用Mock框架模拟Connection类是不是也可以呢？理论上，的确可以。但是想要让ArticleService的测试通过，就必须打开DBArticleRepository的实现，只有配合着其中的实现，才可能让ArticleService跑起来。显然，你跑远了。

现在，对象的创建已经分离了出去，但还是要要有一个地方完成这个工作，最简单的解决方案自然是，把所有的对象创建和组装在一个地方完成：

```
...
ArticleRepository repository = new DBArticleRepository(connection);
AriticleService service = new ArticleService(repository);
...
```

相比于业务逻辑，组装过程并没有什么复杂的部分。一般而言，纯粹是一个又一个对象的创建以及传参的过程，这部分的代码看上去会非常的无聊。

虽然很无聊，但这一部分代码很重要，最好的解决方案就是有一个框架把它解决掉。在Java世界里，这种

组装一堆对象的东西一般被称为“容器”，我们也用这个名字。

```
Container container = new Container();
container.bind(Connection.class).to(connection);
container.bind(ArticleReposistory.class).to(DBArticleRepository.class);
container.bind(ArticleService.class).to(ArticleService.class)

ArticleService service = container.getInstance(ArticleService.class);
```

至此，一个容器就此诞生。因为它解决的是依赖的问题，把被依赖的对象像药水一样，注入到了目标对象中，所以，它得名“依赖注入”（Dependency Injection，简称 DI）。这个容器也就被称为DI容器了。

至此，我简单地给你介绍了DI容器的来龙去脉。虽然上面这段和Spring DI容器长得并不一样，但其原理是一致的，只是接口的差异而已。

事实上，这种创建和组装对象的方式在当年引发了很大的讨论，直到最后Martin Fowler写了一篇《反转控制容器和依赖注入模式》的文章，才算把大家的讨论做了一个总结，行业里总算是有了一个共识。

那段时间，DI容器也得到了蓬勃的发展，很多开源项目都打造了自己的DI容器，Spring是其中最有名的一个。只不过，Spring并没有就此止步，而是在这样一个小内核上面发展出了更多的东西，这才有了我们今天看到的庞大的Spring王国。

讲到这里，你会想，那这和我们要讨论的“模型”有什么关系呢？

正如我前面所说，很多人习惯性把对象的创建和组装写到了一个类里面，这样造成的结果就是，代码出现了大量的耦合。时至今日，很多项目依然在犯同样的错误。很多项目测试难做，原因就在于此。这也从另外一个侧面佐证了可测试性的作用，我们曾在第3讲中说过：可测试性是衡量设计优劣的一个重要标准。

由此可见，在没有DI容器之前，那是怎样的一个蛮荒时代啊！

有了DI容器之后呢？你的代码就只剩下关联的代码，对象的创建和组装都由DI容器完成了。甚至在不经意间，你有了一个还算不错的设计：至少你做到了面向接口编程，它的实现是可以替换的，它还是可测试的。与之前相比，这是一种截然不同的思考方式，而这恰恰就是DI容器这个模型带给我们的。

而且，一旦有了容器的概念，它还可以不断增强。比如，我们想给所有与数据库相关的代码加上时间监控，只要在容器构造对象时添加处理即可。你可能已经发现了，这就是 AOP（Aspect Oriented Programming，面向切面编程）的处理手法。而这些改动，你的业务代码并无感知。

Spring的流行，对于提升Java世界整体编程的质量是大有助益的。因为它引导的设计方向是一个好的方向，一个普通的Java程序员写出来的程序只要符合Spring引导的方向，那么它的基本质量就是有保障的，远超那个随意写程序的年代。

不过，如果你不能认识到DI容器引导的方向，我们还是无法充分利用它的优势，更糟糕的是，我们也不能太低估一些程序员的破坏力。我还是见过很多程序员即便在用了Spring之后，依然是自己构造对象，静态方法满天飞，把原本一个还可以的设计，打得七零八落。

你看，通过上面的分析，我们知道了，只有理解了模型设计的来龙去脉，清楚认识到它在解决的问题，才能更好地运用这个模型去解决后面遇到的问题。如果你是这个项目的维护者，你才能更好地扩展这个模型，以便适应未来的需求。

总结时刻

今天，我们学习了如何了解设计的第一部分：看模型。**理解模型，要知道项目提供了哪些模型，这些模型都提供了怎样的能力**。但还有更重要的一步就是，**要了解模型设计的来龙去脉**。这样，一方面，可以增进了我们对它的了解，但另一方面，也会减少我们对模型的破坏或滥用。

我以Spring的DI容器为例给你讲解了如何理解模型。DI容器的引入有效地解决了对象的创建和组装的问题，让程序员们拥有了一个新的编程模型。

按照这个编程模型去写代码，整体的质量会得到大幅度的提升，也会规避掉之前的许多问题。这也是一个好的模型对项目起到的促进作用。像DI这种设计得非常好的模型，你甚至不觉得自己在用一个特定的模型在编程。

有了对模型的了解，我们已经迈出了理解设计的第一步，下一讲，我们来看看怎样理解接口。

如果今天的内容你只能记住一件事，那请记住：**理解模型，要了解模型设计的来龙去脉。**

思考题

最后，我想请你思考一个问题，DI容器看上去如此地合情合理，为什么在其他编程语言的开发中，它并没有流行起来呢？欢迎在留言区写下你的思考。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- Jxin 2020-06-03 03:01:17

  1.斟酌再三，虽说直接说spring di容器好像也没啥毛病，但个人觉得这描述并不是很准确，故阐述下自己的认知。

  2.我认为spring提供的这个编程模型应该叫ioc（控制反转和响应式编程有点像）而不是di。因为最开始被提出来的是ioc（好莱坞原则），而且最早的实现也不是spring，jdk和ejb都有对ioc的实现，spring才是后来者。但是di确实好像是在spring上被流行起来，且长期主流的（spring di容器没毛病的原因）。不过spring对ioc的实现，除了di还有依赖查找，在我眼里ioc是模型，依赖查找和依赖注入是功能，所以我认为应该是spring提供了ioc的编程模型，利用ioc容器+di的功能简化了开发。

3.依赖注入相对于依赖查找，透明度更好，调用方对ioc容器的api和具体接口实现的查表获取被隐藏了（技术与业务的解耦最终都该透明无感）。但依赖查找在需要动态选择策略时依旧有其用武之地。

4.回答课后题： 对于py和go这类函数式编程语言，函数是一等公民，是可以作为参数传递的。那么直接改变所传的函数就可以实现mock和函数替换（使用和创建天然解耦）。 为什么java会流行？我认为有个原因，是因为java是单分派的语言，编译期方法和参数类型是绑定死的（强类型），运行期走哪个bean的方法是动态决定的。如此就引出了面向接口编程的多态实现方案，才会有后面ioc的诉求。 [16赞]

- 刘丹 2020-06-03 01:03:42
container.bind(Connection.class).to(connection);
container.bind(ArticleReposistory.class).to(DBArticleRepository.class);
container.bind(ArticleService.class).to(ArticleService.class)
请问这3行代码的具体含义是啥？ [3赞]

作者回复2020-06-03 19:05:37
将 Connection 这个类绑定在 connection 这个对象上，当需要一个 Connection 对象时，返回 connection 这个对象。

将 ArticleReposistory 这个接口绑定在 DBArticleRepository 这个类上，当需要一个 ArticleReposistory 对象时，返回 DBArticleRepository 这个类的一个对象。

将 ArticleService 这个类就绑定在其自身，当需要一个 ArticleService 对象时，返回这个类的一个对象。

- Geek_3b1096 2020-06-03 22:22:38
你只是在了解别人设计的结果... 这就是我最欠缺的
[2赞]

作者回复2020-06-04 07:00:28
往前一步，你就成长了。

- 业余爱好者 2020-06-03 13:00:18
一个框架的流行根本原因不是它简化了开发，而是导致了问题的简化的那个开发模型。像spring 提供的di模型，你甚至感受不到它的存在。它更像是一种理念，而这是一个模型的最高级形态。在di的核心模型之上，又出现了starter,auto configuration 等理念，这就是spring boot 的模型创新。在springboot 之上，又有springcloud.....

Spring 这个框架，真的是，，，牛逼(找不到合适的词了） [2赞]

作者回复2020-06-04 07:00:02
没错，简化开发是结果，模型才是动因。

Spring Boot 是在 Spring 出现好多年之后才出现的。

- 捞鱼的搬砖奇 2020-06-03 11:00:37
文中说“静态方法满天飞”是为了在实例方法中调用别的方法所以改为静态方法，是这样的意思吗 [2赞]

作者回复2020-06-03 18:55:49
为了方便，定义静态方法，到处调用，然后，没法 mock，不好测试。

- 骨汤鸡蛋面 2020-06-03 09:43:52
接触spring 七八年， 一直在学习BeanFactory 和 ApplicationContext 上打转，今天才算对容器这个概念

有一个直觉性的认识，感谢老师！ [2赞]

作者回复2020-06-03 18:56:16
学习一个软件，要从基础模型开始。

- Asanz 2020-06-03 09:23:27
DI是模型？我理解的DI是一种实现，IoC是模型😂 [2赞]

作者回复2020-06-03 18:58:37
当年，IoC、DIP和 DI几个名字争论了好久，最后决定叫了DI，这个几个词确实有很多类似的地方。其实，它们都是设计原则。后面讲设计原则的时候，还会提到DIP的。

- 肥宅码农 2020-06-03 08:22:49
我觉得最根本原因是大多数开发不写测试，所以不会考虑依赖问题，大多数方法都是面向实现而不是接口，使用DI容器反而增加了工作量。
目前所在小组偏向于外包，代码只有一层，不是单列，就是静态方法；为了达到快速交付，基本没有设计，不管怎么说这都是不合理的。是前人挖，后人跳。 [2赞]

作者回复2020-06-03 18:59:57
唉，你说的现状，我非常理解。所谓的“快”，只是从当前一个时点上看，放在长期，就是越跑越慢。

- 阳仔 2020-06-03 06:25:14
理解软件设计中模型首先要理解模型解决的核心问题是什么，然后抽丝剥茧了解模型的来龙去脉，深入理解模型解决问题的过程。
spring中的di模型是为了解决对象的创建和组装的问题。
那为什么创建对象和组装要用di来解决?
一个重要的原因是为了解耦。分离接口与实现的强依赖，也就是软件设计第一步分离关注点。
而这个恰恰就是为了可测试性，当一个代码是可测的，其实就是说明它是比较灵活的，修改起来不会牵一发而动全身，提高开发的体验，减少因修改引入的额外问题 [2赞]

- Kǎfkǎ²⁰²⁰ 2020-06-03 05:34:50
多半因为Java在企业级应用里独占鳌头，所以Java的DI更为人所知，也因为更早地出现了容器级的DI，Java才这么流行 [2赞]

作者回复2020-06-03 19:01:57
DI没有成为主流时，Java也已经很流行了，比如，J2EE。

- escray 2020-06-03 11:31:19
其实很早就听说过 Inversion of Control 和 Dependency Injection，但是似乎一直没有搞明白其中的概念，也没有机会有意识的去使用 DI（也许是用了，但是没有意识到）。

重读了 Martin Fowler 的长（旧）文，有一个疑惑，专栏里面的 Spring DI 是属于哪一种类型的 IoC，看上去比较像 type 1，Constructor Injection。

但是在 Martin Fowler 的文章里面说道 Spring 的开发者更推荐使用 Setter Injection（Spring 框架应该是同时支持这两种依赖注入方式的），不知道是因为框架的进展，改用了 Constructor Injection，或者只是局限于作者的这个例子。

结合专栏的内容，简单的了解了一下 Spring 中的 DI。

在 Ruby 中可以使用 dry-rb 实现依赖倒置 https://medium.com/@Bakku1505/introduction-to-depend

ency-injection-in-ruby-dc238655a278

但是 DHH 也说过，Dependency injection is not a virtue https://dhh.dk/2012/dependency-injection-is-not-a-virtue.html [1赞]

- JohnnyB0Y 2020-06-03 09:42:09
  回答作业：
  1，Java有反射，其他语言不一定有；
  2，Java生态比较完善，大神比较多，有模版可以学；
  3，前端开发集中在UI界面和数据解析，需求变更快，用DI容器去做有点吃力；（UI大多是包含的方式，很难把子控件拎出来初始化）
  4，DI容器的AOP可能更适合后端，突如其来的统计、归档之类的需求。而前端的应用生命周期和页面生命周期都由UI框架提供了，AOP自然用的少。

  总结：我觉得AOP可能才是开发者爱用DI的主要原因，加上Java生态的繁荣最终流行起来。（个人看法）[1赞]

- 宝宝太喜欢极客时间了 2020-06-03 08:27:05
  还是感觉模型是个很虚得东西，只可意会不可言传，能不能通过一种方式把他表现出来呢？比如图形？ [1赞]

- 王智 2020-06-04 21:10:46
  读第二遍：模型和模型设计，按照上面所说，Spring DI容器就是一个模型，那好的模型是不是就可以说是一个模型设计呢？按照现在的理解，模型确实不是早期脑海中的类了，早期就觉得类就是模型，从现在看看远远不足呀，模型的具体定义有点模糊了，希望能一直跟下去，看完之后再过一边可能会学到更多的东西。

- 王智 2020-06-04 20:55:35
  我觉得是因为Java面向对象，更多的是使用组合解决问题，使用组合那就避免不了对象的依赖，加上接口实现分离，就更加依赖于DI，而像其他的语言，像面向过程全程使用函数来解决问题，貌似有点用不到对象的组合和创建。我的一点小理解，也不知道有没有问题。

  作者回复2020-06-05 07:40:34
  面向对象和面向过程只是用到了不同的设计元素，其实，使用程序设计语言完全可以兼顾二者，稍后，我会在编程方范式部分进一步讲解。

- 不记年 2020-06-04 14:27:45
  郑老师你好,文中模型感觉和我理解的模型不一样, 我觉得模型是软件内在的东西,是其独有的东西,是对问题域的建模.
  拿spring DI来说,问题域是如何自动化对象的创建和组装
  对问题域建模后,我们的模型由哪几部分组成,各部分之间怎么交互的,我认为这个是模型.至于文中提到的编程模型,我觉得理解成接口更加合适.

- 小鱼儿 2020-06-04 13:08:40
  放下历史长河之中去看问题，比如 现在去看几年前甚至10年前的代码，才知道这样做的好处，分离关注点，可测试性是多么需要，不然真的改不动。

  作者回复2020-06-05 07:58:55
  这是我在开篇词里的立论，软件设计是一门关注长期变化的学问。

- 任旭东 2020-06-04 02:16:16
  为什么不建议使用静态方法？如果只是简单的模型转换，用静态方法不是更好吗？

  作者回复2020-06-04 13:43:53
  静态方法，没法去模拟它的行为，所以，要做测试的话，遇到静态方法，你必须关注它的实现，而不是它的接口。总的来说，静态方法是写着爽，但测着不方便。

- 再来二两杜康酒 2020-06-03 22:15:14
  我觉得可能是用的太浅吧，或者把发挥的空间就留给了一小部分人，所谓的上手快，形成的优质案例不多影响也不大。编程思想是跨语言层面的，比如说java实现的用其它高级语言也能实现，要我说都是用的人的问题。人的因素很大，比如C++也可以不写成多继承啊，不能因为语言层面没有禁止就说是语言设计的不好是吧～